

# A Graphical Language for Proof Strategies

Gudmund Grov<sup>1</sup> and Aleks Kissinger<sup>2</sup>

<sup>1</sup> School of Mathematical and Computer Sciences, Heriot-Watt University,  
Edinburgh, UK, [G.Grov@hw.ac.uk](mailto:G.Grov@hw.ac.uk)

<sup>2</sup> Department of Computer Science, University of Oxford, UK,  
[aleks.kissinger@cs.ox.ac.uk](mailto:aleks.kissinger@cs.ox.ac.uk)

**Abstract.** Complex automated proof strategies are often difficult to visualise, modify, and debug. Traditional tactic languages, often based on stack-based goal propagation, make it easy to write proofs that obscure the flow of goals between tactics and are fragile to minor changes in input or proof structure. Here, we address this by introducing a graphical language for writing proof strategies. Strategies are constructed visually by “wiring together” collections of tactics and evaluated by propagating goal nodes through the diagram via graph rewriting. Tactic nodes can have many output wires, and use a filtering procedure based on goal-types (predicates describing the features or “shape” of a goal) to decide where best to send newly-generated subgoals. In addition to making the flow of goal information explicit, the graphical language can fulfil the role of many tacticals using visual idioms like branching, merging, and feedback loops. We argue that this language enables development of more robust proof strategies and provide a prototype implementation in Isabelle.

## 1 Introduction

In existing tactic languages for proofs, it is often difficult to compose tactics in such a way that all goals are sent to the correct target, especially when different goals should be handled differently. Moreover, when a large tactic fails, it is hard to analyse where the failure occurred during debugging. If the structure of a tactic is difficult to understand, often the easiest way for a user to deal with failure is to manually guide the proof until the tactic succeeds (or becomes unnecessary), rather than correcting the weakness of the tactic itself. In this case, the proof is made more complicated and insight from this failure is not carried across to other proofs. Thus, a tactic language where it is easy to diagnose and correct failures will lead to better tactics and simpler, more general proofs.

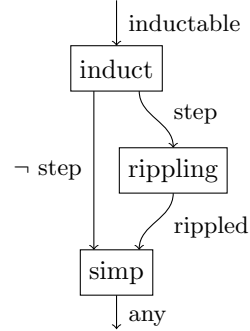
Firstly, we argue that many errors could have been found statically. The problem is that *tactics* are essentially untyped: they are functions from a *goal* to a set of possible sub-*goals*. In many programming languages, *types* are used statically to rule out many “obvious” errors. For example, in (typed) functional languages, a type error will occur when one tries to compose two functions which do not have a unifiable type. In an untyped tactic language, this kind of “round-peg-square-hole” situation will not manifest until run-time.

Without a means of distinguishing goals, tactics have no choice but to rely on the order in which goals arrive. Small changes to tactics can lead to failures. For example, consider a case where we expect three outputs of tactic  $t_1$ , where the first two are sent to  $t_2$  and the last to  $t_3$ . A small improvement of  $t_1$  may result in only two sub-goals. This “improvement” causes  $t_2$  to be applied to the second goal when it should have been  $t_3$ . The tactic  $t_2$  may then fail or create unexpected new sub-goals that cause some later tactic to fail.

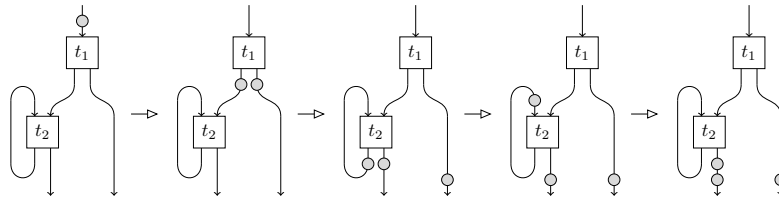
Finally, for errors that cannot be found statically, it is very hard to inspect and analyse the failures during debugging. In the above example, if  $t_2$  creates sub-goals that tactics later in the proof do not expect, the error will be reported in a completely different place. Without a clear handle on the flow of goals through the proof, finding the real source of the error could be very difficult indeed.

In this paper, we address these issues by introducing a graphical proof strategy language. The primary aims of the language are (i) to improve robustness of proof strategies with static goal typing and type-safe tactic “wirings” and (ii) to improve the ability to dynamically inspect, analyse, and modify strategies, especially when things go wrong.

The language is expressed using *string diagrams*, which provide a general way of describing composed processes. They originated with Penrose [16] as a means of describing tensor contractions, but were later shown by Joyal and Street to be useful in a much more general context [13]. In our graphical language, tactics are represented by boxes and wires are labelled with goal-types. This is illustrated on the right for a variant of the rippling strategy [5], which guides rewriting of the (conclusion of the) induction step case towards the induction hypothesis.



Evaluation is done by propagating goal nodes toward the output wires using graph rewriting.



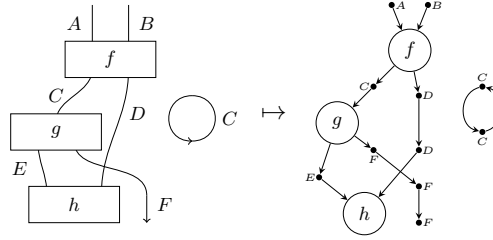
To define and manipulate string diagrams programmatically, we use a combinatoric representation of string diagrams called *string graphs* [8]. String graphs admit type-safe composition via pushout and type-safe transformation via double-pushout (DPO) graph rewriting. We provide a brief (non-categorical) background on string diagrams, string graphs, and string graph rewriting in section 2.

After introducing goal-types and typed tactics in section 3, we describe the language and evaluation procedure for strategy graphs in section 4. In section 5

we enrich this language with combinators and a notion of hierarchical evaluation. A prototype implementation in Isabelle is described in section 6 with examples in section 7. We discuss related and future work in sections 8 and 9, respectively.

## 2 Background: String Diagrams and String Graphs

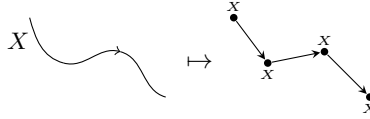
The graphical proof strategy language is built on the mathematical formalism of *string graphs*. String graphs were defined in [8] (called open-graphs therein) to provide a rigorous rewrite theory for *string diagrams*.



**Fig. 1.** A string diagram and its associated string graph

A string diagram consists of *boxes* and *wires*. A box represents some kind of operation with multiple (distinguished) inputs and outputs, and wires express the composition of these boxes. Note that, unlike graph edges, these wires are allowed to be *free* (i.e. not connected to a box) at one or both ends. These free wires are used to express inputs and outputs of the diagram (see LHS of Figure 2). Note that wires can even be connected to themselves to form circles. While this is an important feature of the general theory of string diagrams, circles will not be relevant for proof strategy graphs.

String diagrams can be formalised as topological graphs with some extra structure. While intuitive, they are unwieldy if one wishes to develop a mechanised rewrite theory. To solve this problem, *string graphs* were defined to represent string diagrams as “plain old” typed graphs with a well-defined notion of substitution: namely the *double-pushout* approach to graph rewriting. The main trick is to replace wires with chains of special vertices called *wire vertices*.



We also introduce *node-vertices*, which represent “logical” nodes (aka boxes) in the diagram.

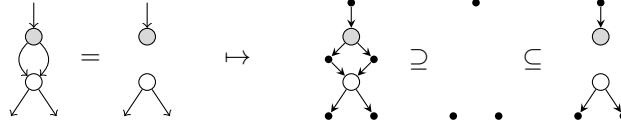
We parametrise string graphs by a signature  $T$ , which defines a set of nodes and the types of their inputs and outputs.  $T$  consists of a set  $\mathcal{M}$  of *maps*, a set  $\mathcal{O}$  of types, as well as functions  $\text{dom}, \text{cod} : \mathcal{M} \rightarrow [\mathcal{O}]$  that assign input and output types for each map, where  $[\mathcal{O}]$  represents lists of elements of  $\mathcal{O}$ .

Henceforth, we will refer to chains of wire vertices simply as wires, when there is no ambiguity. There are a few things to note here. Firstly, wire-vertices carry

the type of a wire, so every connection between two node-vertices must contain at least one node-vertex. We do not allow wires to split or merge, so wire-vertices must have at most one input and one output. If nodes are not symmetric in their inputs or outputs, we distinguish inputs and outputs using edge types ( $\text{in}_1, \text{in}_2, \text{out}_1, \text{out}_2, \dots$ ).

**Definition 1.** If a wire-vertex has no in-edges, it is called an *input*. We write the set of inputs of a string graph  $G$  as  $\text{In}(G)$ . Similarly, a wire-vertex with no out-edges is called an *output*, and the set of outputs is written  $\text{Out}(G)$ . The inputs and outputs define a string graph’s *boundary*, written  $\text{Bound}(G)$ . The set of vertices and edges not in the boundary of a string graph is called its *interior*.

In [8], the authors showed that string diagrams can be faithfully represented using string graphs, and that equations between string diagrams can be represented by *string graph rewrite rules*.



Note that in a string diagram equation, the input and output arities on the LHS and RHS always match. Also, we have implicitly stated which inputs/outputs on the LHS correspond to which inputs/outputs on the RHS. We make this explicit in the string graph picture by requiring that the LHS and the RHS *share the same boundary*.

**Definition 2.** A *string graph rewrite rule*  $L \twoheadrightarrow R$  is a pair of graphs that share the same boundary, i.e.  $\text{In}(L) = \text{In}(R)$  and  $\text{Out}(L) = \text{Out}(R)$ .

It is often convenient to encode this correspondence as a span of injective graph homomorphisms  $G \leftarrow B \rightarrow H$ , in which case the boundaries need not be equal on-the-nose, but merely isomorphic as typed graphs. The site of a rewrite is marked by a *matching*. This is an injective graph homomorphism  $m : L \rightarrow G$  with the additional property that edges in the rest of  $G$  may only connect to  $L$  via its boundary. In other words, there must be no edges connecting the interior of  $m(L)$  to  $G - m(L)$ .

We can now define rewriting in terms of a matching. Given a rule  $L \twoheadrightarrow R$  and a matching  $m : L \rightarrow G$ :

1. Rename  $L \twoheadrightarrow R$  to an equivalent rule  $L' \twoheadrightarrow R'$  such that  $L' = m(L)$  and the vertices/edges in the interior of  $R'$  are fresh in  $G$ .
2. Let  $G'$  be  $G$  with the interior of  $L'$  removed.
3. Form the rewritten graph as  $G' \cup R'$ .

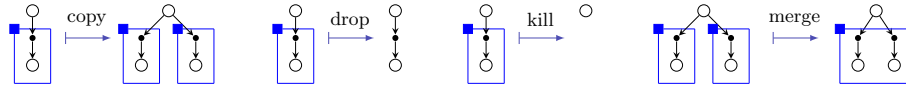
Since  $\text{Bound}(L') (= \text{Bound}(R'))$  is never removed, it provides the “glue” for attaching  $R'$  to the new graph. In categorical terms, step 2 is a *pushout-complement* and step 3 is a *pushout*. As the whole process involves two pushouts, this is called the *double-pushout approach* to graph transformation [9].

For our purposes, we also allow vertex and edge data to carry free variables. For this case, in step 1 we additionally perform substitutions of free variables to instantiate  $L' \multimap R'$ .

Often it is convenient to express families of string diagrams that have some repeated structure. When speaking about such families informally, it suffices to use ellipses to suggest repetition.



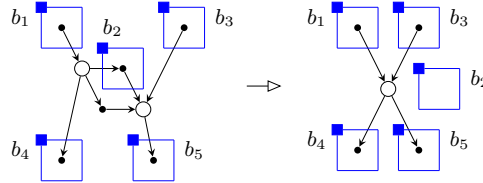
However, when it comes to formalising these diagrams as families of string graphs or string graph rewrite rules, it is useful to be more precise. For this, we introduce the notion of a *!-box* (pronounced “bang-box”), which can be instantiated by performing a sequence of the following operations.



A graph containing *!-boxes* is called a *!-graph*. The family of concrete graphs represented by a *!-graph* are precisely those that can be obtained by performing any of the 4 *!-box* operations zero or more times.

$$\left[ \begin{array}{c} \bullet \\ \downarrow \\ \boxed{x} \end{array} \right] = \left\{ \bullet, \begin{array}{c} \bullet \\ \downarrow \\ x \end{array}, \begin{array}{c} \bullet \\ \swarrow \searrow \\ x \quad x \end{array}, \begin{array}{c} \bullet \\ \swarrow \downarrow \searrow \\ x \quad x \quad x \end{array}, \dots \right\}$$

We can represent infinite families of rewrite rules as a pair of *!-graphs* with corresponding *!-boxes* satisfying certain conditions to ensure that all concrete rules are valid.



Like in the graph case, rules can be instantiated by performing the four *!-box* operations, with the additional condition that whenever an operation is applied to a *!-box* on the LHS, the same operation is applied to the corresponding *!-box* on the RHS. For more details on the definition and semantics of *!-graphs*, along with *!-graph* matching and rewriting, see [15].

Normally, when a *!-box* is copied the data on vertices and edges are copied verbatim. We extend the *!-box* language slightly by allowing free variables inside the box to be bound as *fresh names* using a  $\phi$  followed by the variable name. When a *!-box* is copied, any variables marked as fresh will be replaced with new, fresh free variables in the new copy.

$$\left[ \begin{array}{c} \bullet \\ \downarrow \\ \boxed{\phi x} \\ \downarrow \\ x \end{array} \right] = \left\{ \bullet, \begin{array}{c} \bullet \\ \downarrow \\ x' \end{array}, \begin{array}{c} \bullet \\ \swarrow \searrow \\ x' \quad x'' \end{array}, \begin{array}{c} \bullet \\ \swarrow \downarrow \searrow \\ x' \quad x'' \quad x''' \end{array}, \dots \right\}$$

### 3 Goal Types and Typed Tactics

For a type  $\tau$ , let  $[\tau]$  be the type of finite lists and  $\{\tau\}$  be the type of finite sets whose elements are of type  $\tau$ .

Rather than considering all goals as members of one big type “**goal**”, assume that we have a partially ordered set of goal types  $\mathcal{G}$ . A particular goal type  $\alpha \in \mathcal{G}$  represents all goals with some particular features, which may include local properties like “contains symbol  $X$ ”, global properties like shared meta-variables, and tracing properties such as features of the parent goal. Since types are essentially predicates defined over goals, we assume the ability to take arbitrary meets and joins of goal types.

**Definition 3.** For  $\alpha \wedge \beta$  the meet of two goal types and  $\perp$  the empty goal type, we say  $\alpha$  and  $\beta$  are *orthogonal* if  $\alpha \wedge \beta = \perp$ .

**Definition 4.** A typed tactic is a function of the form:

$$\mathbf{ttac} : \alpha \rightarrow \{[\beta_1] \times [\beta_2] \times \dots \times [\beta_n]\}$$

where  $\alpha$  and  $\beta_i$  are goal types, for all  $i$ .

For our purposes, the usual notion of a tactic can be treated as a function of the form:

$$\mathbf{tac} : \mathbf{goal} \rightarrow \{[\mathbf{goal}]\}$$

That is, it takes a single goal to a set whose elements are lists of subgoals (and possibly some other data such as a justification function) representing each evaluation of the (non-deterministic) tactic.

For a given input type  $\alpha$  and a sequence  $\beta_i$  of output types, we can lift  $\mathbf{tac}$  to a typed tactic  $\mathbf{tac}^* : \alpha \rightarrow \{[\beta_1] \times [\beta_2] \times \dots \times [\beta_n]\}$  as follows. First, for a goal list  $L$ , let  $p(\beta_1, \dots, \beta_n; L)$  be the set of all partitions of  $L$  into  $n$  lists  $\{L_1, \dots, L_n\}$ , where all of the goals in the  $i$ -th list have goal type  $\beta_i$ . Then:

$$\mathbf{tac}^*(g) = \begin{cases} p(\beta_1, \dots, \beta_n; \mathbf{tac}(g)) & \text{if } g \text{ is of type } \alpha \\ \emptyset & \text{otherwise} \end{cases} \quad (1)$$

If we disregard the order of goals in each list  $L_i$ , this can be done without introducing non-determinism, provided the types  $\beta_i$  are all orthogonal. Tactics of the above form are the simplest constituents of strategy diagrams.

**Definition 5.** Typed tactics of the form of equation (1) are called *atomic tactics*.

We shall see examples of tactics that are not atomic later, when we discuss hierarchies. Henceforth, we will refer to typed tactics simply as tactics.

## 4 Definition and Evaluation of Proof Strategy Graphs

It is possible to represent a proof strategy as a string diagram, where wires are labelled with goal types.

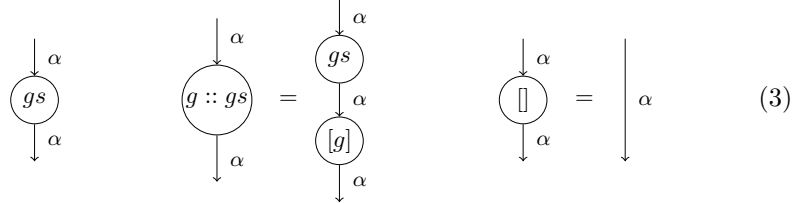
Proof strategy diagrams contain three kinds of nodes. The first are tactic nodes, which we shall represent as boxes.



The second type of node is a *merge node*. These can have any number of inputs and one output. They are symmetric in their inputs, and adjacent merge nodes can be combined.



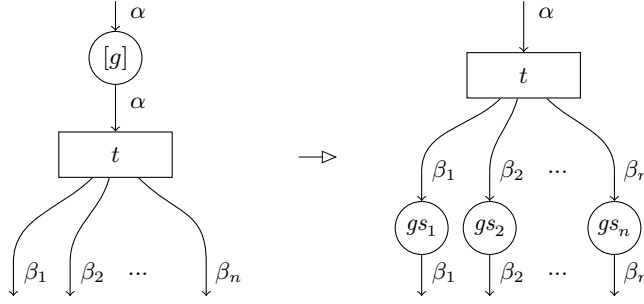
These two kinds of nodes suffice to define a proof strategy. Proof strategies have a “token-style” evaluation semantics, where goals move along wires and are processed by tactics. To model this evaluation using diagram rewrites, we introduce explicit goal nodes. These always have one input and one output wire labelled by a goal type  $\alpha$ . The node itself contains a list of goals of type  $\alpha$ . Whereas tactic nodes produce lists of goals, they only take one goal at a time as input<sup>3</sup>. Therefore, we introduce identities for reducing lists to singletons.



Evaluation of a strategy graph proceeds via the **eval** meta-rule (Fig. 4). This rule acts locally on a tactic and a goal singleton by applying the tactic on propagating subgoals to the relevant output wires. First, we’ll show the case of a single input wire. Evaluation occurs in four steps:

1. match the LHS of the **eval** rule, partially instantiating the rule
2. evaluate the tactic function for the matched input and output types
3. finish instantiating the RHS with the lists  $gs_i$  produced by the tactic
4. apply the fully instantiated rule(s)

<sup>3</sup> The use of goal-list and singleton lists for evaluation simplifies the evaluation semantics (compared with single goal nodes or evaluation of arbitrary sized goal-list).



**Fig. 2.** The **eval** meta-rule

First, **eval** is matched as usual by finding substitutions for  $\alpha$ ,  $\{\beta_i\}$ ,  $t$ , and  $g$ . At this point, the resulting rule  $L \rightarrow R$  is only *partially* instantiated, in that there are still free variables  $\{gs_i\}$  in the RHS. To finish instantiating the rule, we apply the associated typed tactic function. Suppose  $t = \mathbf{tac}$ , we then lookup a function called **tac** with the signature:

$$\mathbf{tac} : \alpha \rightarrow \{[\beta_1] \times [\beta_2] \times \dots \times [\beta_n]\}$$

Tactics need not be defined for every type signature. If the matched tactic is not defined, simply return the fail function  $\mathbf{fail}(g) = \{\}$ .

Next, we can apply the tactic to get a (lazy) set of evaluations  $E = \mathbf{tac}(g)$ . For each evaluation  $(gs'_1, \dots, gs'_n) \in E$ , we can instantiate a rewrite rule  $L \rightarrow R'$  by substituting  $gs'_i$  for  $gs'_i$  in  $R$ .

Thus, for each evaluation of a given tactic, we get a matching fully-instantiated rewrite rule  $L \rightarrow R'$  and a matching  $m : L \rightarrow G$  on the strategy graph.

*Example 1.* Consider the conjecture  $\mathit{even}(2*n)$ . This can be proved by induction with two base cases. Suppose the conjecture is on the input wire of a tactic node labelled with: **induct** : **inductable**  $\rightarrow \{[\neg\mathbf{step}] \times [\mathbf{step}]\}$ . This is then evaluated as follows. In step 1, the variables are (partly) instantiated to  $\sigma_1$ :

$$\sigma_1 = \{\alpha \mapsto \mathbf{inductable}, \beta_1 \mapsto \neg\mathbf{step}, \beta_2 \mapsto \mathbf{step}, t \mapsto \mathbf{induct}, g \mapsto \mathit{even}(2*n)\}.$$

Next, the underlying (untyped) induct tactic is applied to  $g$  creating the sub-goals:  $\mathit{even}(2*0)$ ,  $\mathit{even}(2*1)$  and  $\mathit{even}(2*n) \vdash \mathit{even}(2*S(n))$ . Since the output types are orthogonal, the lifted tactic will return a set containing a single pair of goal-lists:  $([\mathit{even}(2*0), \mathit{even}(2*1)], [\mathit{even}(2*n) \vdash \mathit{even}(2*S(n))])$ .

In the third step the instantiation of the rewrite rule is completed by

$$\sigma = \sigma_1 \cup \{gs_1 \mapsto [\mathit{even}(2*0), \mathit{even}(2*1)], gs_2 \mapsto [\mathit{even}(2*n) \vdash \mathit{even}(2*S(n))]\}.$$

Finally, the fully-instantiated rule can be applied to propagate the goal node through the tactic.

**Definition 6.** A strategy graph is in *evaluation normal form* (ENF) if the only remaining goal nodes are on output wires of the graph.



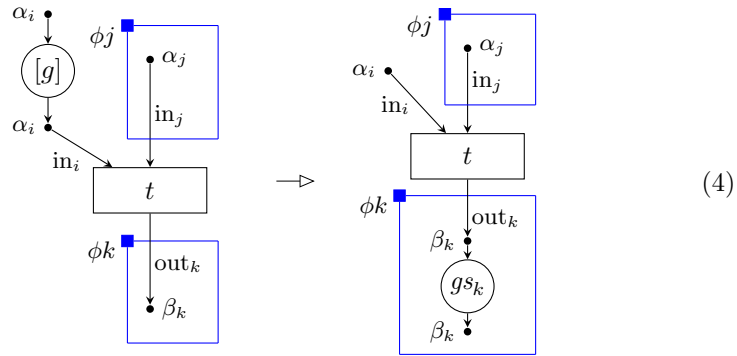
A naïve strategy for using the above procedure to generate graphs in ENF is as follows:

1. use the above procedure to instantiate and apply **eval**
2. unfold any (non-singleton) goal lists using the rules given by (3)
3. repeat until **eval** yields no matches

A graph may contain many goals (possible **eval** matches). If there are no shared properties (e.g. meta-variables) between the goals, the result is independent of the application order modulo the goal order on the output wires. One may want certain *evaluation strategies*, such as ‘evaluate the simplest goal first’ or ‘instantiate meta-variables in a certain order’. More sophisticated evaluation procedures can be achieved by configuring the rewrite engine with such evaluation and search strategy. However, the underlying strategy graph is independent of such details. Note that in the presence of feedback, a strategy is not in general guaranteed to terminate – this has to be proven on a case-by-case basis. If the graph is acyclic, it will always terminate (provided all of the tactic functions do).

We handle tactic boxes with multiple input wires similarly, in that only one goal is processed at a time. The only difference is that a tactic can have distinct evaluation functions  $\{\mathbf{tac}_1, \dots, \mathbf{tac}_n\}$ . The function  $\mathbf{tac}_i$  is invoked when the matched goal node occurs on the  $i$ -th input wire of the tactic node. This position data will be important for graph tactics, introduced in the section 5. For atomic tactics, it is irrelevant, so all of the functions  $\mathbf{tac}_i$  are taken to be identical.

This evaluation “meta-rule” can be formalised over string graphs using a single  $!$ -graph rewrite rule.



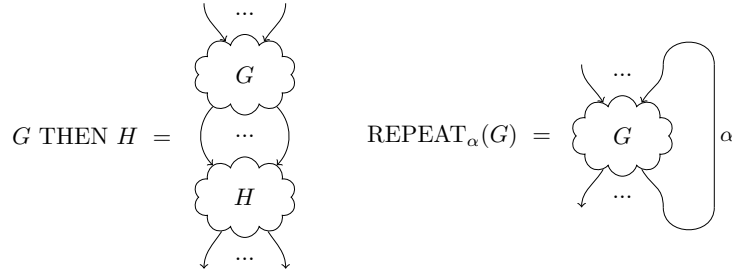
Note how wire vertices carry the goal type information and edge data is used to distinguish the positions of input and output wires on the tactic node. The use of fresh-name binders on the indices  $j$  and  $k$  is crucial here to allow them to match distinct input and output edges.

## 5 Combinators and Hierarchies

An important aspect of any tactic language is the mechanism by which existing tactics can be combined.. Strategy graphs already admit a natural interpretation

for many familiar tacticals via graph combinators. For instance, if a strategy graph  $G$  has output wires with type  $\beta_1, \dots, \beta_n$  and another strategy graph  $H$  has input wires of the same type, the strategy graph  $(G \text{ THEN } H)$  can be obtained by plugging all of the outputs of  $G$  into all of the inputs of  $H$ .

Suppose a strategy graph has  $\alpha$  as input and output type. Then, we could introduce a feedback loop, or “trace” in categorical terminology, that will send goals of type  $\alpha$  back to the input. Such a REPEAT combinator generalises both REPEAT\_WHILE and REPEAT\_UNTIL. Furthermore, if  $\alpha$  captures exactly those goals for which the strategy graph will not fail, this becomes the traditional REPEAT tactical.



We can also define a much richer notion of composition based on *graph tactics* that enables us to write hierarchical proof strategies (in the spirit of, e.g. Hi-Tacs [2,17]) and elegantly express branching strategies using OR and ORELSE combinators.

**Definition 7.** A *graph tactic* is a typed tactic

$$G\text{-}\mathbf{tac}_i : \alpha_i \rightarrow \{[\beta_1] \times [\beta_2] \times \dots \times [\beta_n]\}$$

where  $G$  is a strategy graph whose  $i$ -th input is of type  $\alpha_i$  and whose output types are  $\beta_1, \dots, \beta_n$ .

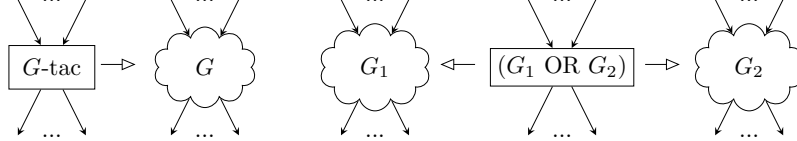
For a goal  $g$ , the evaluation  $G\text{-}\mathbf{tac}_i(g)$  is computed as follows:

1. add  $[g]$  to the  $i^{\text{th}}$  input wire of  $G$
2. use the evaluation rule (4) to produce zero or more graphs  $G'$  in ENF
3. for each  $G'$ , return a tuple  $(gs_1, \dots, gs_n)$ , which  $gs_j$  is the list of all goals remaining on the  $j^{\text{th}}$  output edge of  $G'$

A common LCF tactical is the OR combinator, where  $\text{OR}(\mathbf{tac}_1, \mathbf{tac}_2)$  either applies tactic  $\mathbf{tac}_1$  or tactic  $\mathbf{tac}_2$ . We can lift this tactical to the level of proof strategy graphs, by evaluating both graphs and taking the union of the resulting evaluations. ORELSE is similar, except it will short-circuit the second tactic if the first succeeds. For two graphs  $G$  and  $H$  with the same input and output types, we define the following typed tactics for OR and ORELSE.

$$\begin{aligned} (G \text{ OR } H)_i(g) &:= G\text{-}\mathbf{tac}_i(g) \cup H\text{-}\mathbf{tac}_i(g) \\ (G \text{ ORELSE } H)_i(g) &:= \begin{cases} G\text{-}\mathbf{tac}_i(g) & \text{if } G\text{-}\mathbf{tac}_i(g) \neq \{\} \\ H\text{-}\mathbf{tac}_i(g) & \text{otherwise} \end{cases} \end{aligned}$$

A consequence of the recursive evaluation of nested graphs is that different search and evaluation strategies can be applied to different parts of a proof, by attaching special strategies to graph tactics. One can easily turn a graph tactic into a rewrite rule, and an OR/ORELSE combinator into a set of rewrite rules, by lifting the tactic node into a graph with the same boundary as the graph it encapsulates. This can be useful for interactive usage.



Note that the THEN combinator and OR/ORELSE combinators rely on strict matching for input/output types. One could implement these in a more flexible manner by using typing information to automatically permute inputs and outputs to fit graphs together, where possible. If the relevant types are orthogonal, this can even be done without introducing any non-determinism.

In a longer version of this paper, currently under development, we plan to formalise the combinators outlined here together with several variations of them.

## 6 Implementation

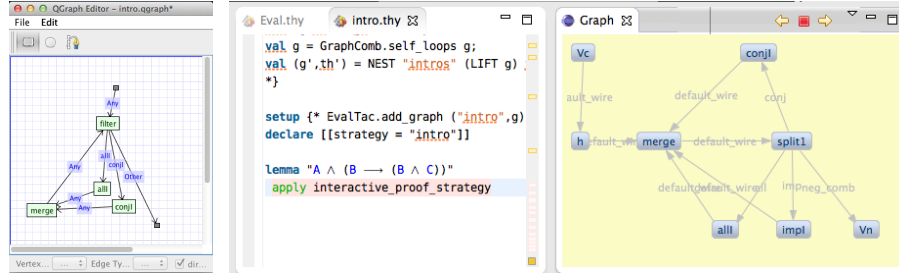
A prototype has been implemented for the Isabelle theorem prover<sup>4</sup>.

**Proof Representation** In Isabelle, all subgoals are stored within a single `thm` type, e.g. if subgoal  $G$  and  $H$  remain to prove the goal  $F$ , then this is stored internally as  $G \implies H \implies F$ . When  $G$  and  $H$  are discharged, then only  $F$  is left and the proof is complete. Inspired by the way Isabelle/Isar handles structured proofs, we have created a proof representation where each subgoal is stored separately. In order to discharge  $G \implies H \implies F$ , having proved  $G$  and  $H$ , we use backwards resolution in an “export phase” which turns our internal representation into a `thm` that can be used as an Isabelle/Isar tactic/method. This integration is still under development, and some features of Isabelle `thm` objects (e.g. schematic variables) are not yet supported.

**Proof Strategy Graph Representation** String graph representation and rewriting is implemented using the Quantomatic [14] library. Among other things, Quantomatic provides functors for defining graph theories, given expressions for vertex and edge data, along with functions for matching and unifying data. The core rewriting engine is implemented in Poly/ML – the implementation language of Isabelle. First, we have combined these two tools, by creating an Isabelle theory which includes the Quantomatic core. On top of that we have implemented the theory described in the previous sections for proof strategies.

In addition to the graph, a separate proof object is used, which keeps track of each goal, and their internal relationship. An atomic tactic contains an Isabelle tactic, while a graph tactic uses a graph stored in the Isabelle context.

<sup>4</sup> See <https://github.com/ggrov/psgraph/tree/itp13>.



**Fig. 3.** Screenshot of GUI. Left: Drawing GUI. Right: Interactive Execution GUI.

Goal types are represented by a name and a set of named *features*. To encode a feature, we have defined a set of *feature types*, which consists of a name and a matching function on the goal, which may take some arguments. A goal then matches a goal type if all the features in the goal type matches. For example, one supported feature is the *top level symbol* of a goal's conclusion. This is represented by a named feature type *top\_level\_symbol*, which accepts the name of the symbol as argument. For example,  $(\wedge, \text{top\_level\_symbol})$  means that  $\wedge$  is the top-level symbols and  $(\text{conj}, \{(\wedge, \text{top\_level\_symbol})\})$  is the **conj** edge in the example in section 7.

The prototype implementation deviates from the theory in that relevant rewrite rules are generated on-the-fly to match the neighbourhood of a given tactic node. This is to overcome the fact that  $!$ -boxes in Quantomatic do not yet support fresh-name binders, as used in the meta-rule at the end of section 4. The current representation also does not distinguish input and output edge order on nodes. However, this has little effect on the strategies tested, as type information typically suffices to direct goals to the appropriate input/output.

**Executing Proof Strategies** Proof strategy graphs can be registered in advance and invoked via an Isabelle method called **proof\_strategy**, which uses a variable **strategy** holding the name of the active strategy. Thus, a strategy graph can be invoked using the following command in an Isabelle/Isar proof mode:

**using**  $[[\text{strategy} = \dots]]$  **apply** **proof\_strategy**

The goals on the output edges of the given strategy graph become new sub-goals after executing this method.

**Implementing Graphs** Strategy graphs can be implemented directly at the ML-level using the following procedure:

1. Implement the *goal-types* and declare I/O types for *tactics*.
2. Lift typed tactics into graphs and combine them into a proof strategy graph using available combinators.
3. Register the graph with a name in the Isabelle theory.

**Drawing Graphs** A more natural way to work with graphs is to draw them in a GUI. After a strategy graph is created in our graph editor (see LHS of Fig. 3), it can be saved as a file (say **intro.qgraph**). This can then be imported to

Isabelle, and is automatically registered with the base name of the file, in this case *intro*. Our ultimate goal is to extend the GUI support so that strategy graphs can be written and modified inline, rather than needing to be continuously saved and re-loaded. One still have to implement goal-types and associate a name for each of the tactics used in this case.

**Debugging Interface** There is also an interactive version of the `proof_strategy` method, called `proof_strategy_interactive`. This method uses a socket to communicate with a GUI written as an Eclipse plug-in – and thus works best with the Isabelle/Eclipse version of the Isabelle PIDE<sup>5</sup>. However, it can still be used from e.g. Isabelle/jEdit. This interface is shown on the right hand side of Figure 3, and contains three buttons: one for back-tracking, one to terminate the proof process; and one for the next step. When terminating a partial proof, the remaining sub-goals become new sub-goals. This interface is mainly used for debugging, testing and analysis of a proof strategy.

## 7 Tool Illustration: an Introduction Proof Strategy

To illustrate the language and the tool, we will use “failure-analysis” to develop a version of the well-known *introduction tactic*. Although this is well-understood and straightforward to implement in standard tactic languages, it is sufficiently simple to illustrate some properties of the language and tool for the space available here.

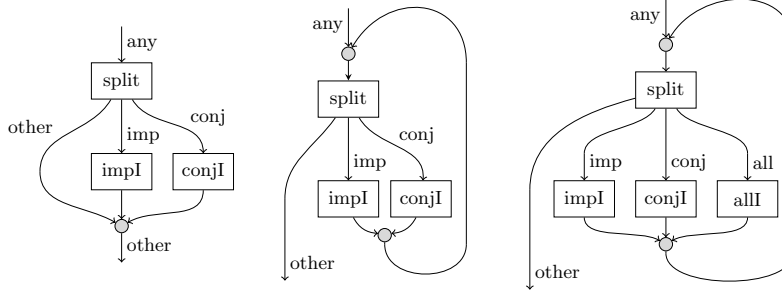
*Example 2 (First Version).* The strategy works on **any** goal, thus we uses this  $\top$  goal-type as an input. Next, we introduce three new goal-types: **imp** which holds for a goal with a conclusion of a logical implication; **conj** which holds for a goal with a conclusion of a logical conjunction; and **other** which “negates” the previous two, that is, it holds for a goal with a conclusion that is neither a conjunction nor an implication. Next, we introduce some atomic tactics:

- **split** : **any**  $\rightarrow$   $\{[\mathbf{imp}] \times [\mathbf{conj}] \times [\mathbf{other}]\}$  is an identity tactic used to filter the goals according to their goal-type.
- **impI** : **imp**  $\rightarrow$   $\{[\mathbf{other}]\}$  is the implication introduction tactic.
- **conjI** : **conj**  $\rightarrow$   $\{[\mathbf{other}]\}$  is the conjunction introduction tactic.

and draw the left most diagram of Fig. 4.

*Example 3 (Second Version).* The strategy works well for goals such as  $A \rightarrow B$  and  $A \wedge B$ . However the tool will fail for goals such as  $A \rightarrow B \wedge C$ , since the output goal of **impI** will be  $B \wedge C$  which fails for the **other** type. Thus, we change the output to **any**, so that **impI** : **imp**  $\rightarrow$   $\{[\mathbf{any}]\}$  and **conjI** : **conj**  $\rightarrow$   $\{[\mathbf{any}]\}$ . Moreover, the intuition behind this strategy is that goals on the output should not have a conclusion of the shape of a conjunction or implication which it will for the goal  $A \rightarrow B \wedge C$ . Thus, we introduce the output from **impI** and **conjI** are fed back into **split**. This is the middle diagram of Fig. 4.

<sup>5</sup> See <https://github.com/andriusvelykis/isabelle-eclipse>.



**Fig. 4.** Intro strategy graphs

*Example 4 (Third Version).* We can also analyse the graph to easily spot a *missing case*. E.g. consider running the tool on the goal  $A \rightarrow B \wedge (\forall x. P x)$ . On termination, the output on the **other** wire will contain the goals  $A$ ,  $B$  and  $\forall x. P x$ . Let's assume that these eventually reaches a tactic which do not support universal quantifiers. By inspecting the execution, one can easily spot where our strategy failed. Thus, we introduce a new goal-type **all**, which succeeds when the conclusion of the goal has the shape of a universal quantifier. Then, we update **split** into **split** : **any**  $\rightarrow \{[\mathbf{imp}] \times [\mathbf{conj}] \times [\mathbf{all}] \times [\mathbf{other}]\}$ , and **other** so it also fails when the conclusion starts with a universal quantifier. The resulting strategy is the right-most diagram of Fig. 4. The same approach can be used to discover the remaining cases of the introduction tactic. Note that if the input type of the target does not support universally quantified goals, then this error would have been found statically during composition (causing evaluation to fail).

## 8 Related Work

We often distinguish between *procedural* proofs, where a proof is described as a sequence of tactic applications (i.e. function composition), ignoring the goals; and *declarative* or *structured* proofs, where the proof is described in terms of intermediate goals (goal islands), and the actual proof commands are seen more as a side issue. In our language, these two views are merged in the sense that the *goal-type* and goals on the wires create a declarative view, while the graph as a whole gives a *procedural* view of how tactics are composed.

Autexier and Dietrich [3] have developed a *declarative tactic language on top of a declarative proof language*, where a proof *schema* is created together with variables that can be instantiated by matching when a particular goal is applied to it. Their work seems closer to the declarative view, whilst we believe that our work is more general in terms of composing smaller strategies.

Similarly, there have been several attempts to create *declarative tactic languages on top of procedural tactic languages* [11,10]. Asperti et al [1] argues that these approaches suffer from two drawbacks: goal selection for multiple sub-goals, and information flow between tactics. We believe that our lifting to goal-types and compositions in terms of them addresses both drawbacks.

The ability to write hierarchical proofs is also supported in some tactic languages, most notably HiTacs [2,17]. By utilising goal types we can write more general hierarchies compared with HiTacs, since we allow multiple input edges

and each input edge may contain multiple goals, whilst HiTacs only allows one input goal for a hierarchy. Like us, the authors develop a general theory for their tactic language, independent of the underlying theorem prover.

The goal types in our language can be seen as a lightweight implementation of pre/post-condition as in *proof planning* [4], with the additional property that the language captures the flow of goals. The work did in fact initially start as a new version of the IsaPlanner proof planner [7], but a proof representation closer to Isabelle turned out to be more practical. In the future we plan to include *proof critics* [5] into the language, which can help automate patches to common failures. In particular, the use of goal- types to constrain the search for missing lemmas.

It is important to note the difference with the field of *diagrammatical reasoning*, as in e.g. [12] and [?], where diagrams are the objects of interest for reasoning rather than the means of capturing the reasoning process.

## 9 Conclusion and Future Work

By lifting proof strategies to the level of goal-types rather than the level of goals, we are able to write more robust strategies, and no longer rely on the number and order of sub-goals resulting from a tactic application for tactic composition. Thus, the problem of goal selection/focus/classification when composing tactics, as highlighted in [1], is significantly improved. Since composition of proof strategies is also at the level of goal-types, we increase type safety and enable better static analysis. The use of graphs to represent the flow of goals further enables visualisation of partially evaluated proofs and improves debugging, failure analysis and failure patching.

Consider, for example, the *rippling strategy* from section 1. In a traditional tactic language, it would have been easy to mistakenly send all goals to *simp* before *rippling* was attempted, which could break the syntactic requirement of rippling. Such mistakes are inherently difficult to find, and we believe that our language greatly simplifies this kind of situation.

We are working on improving the theory to take advantage of the partial order structure to admit sub-typing and type unification on wires. Further, we would like to move the implementation closer to the theory by using built-in !-box mechanisms, rather than generating rules on the fly. We are also working on improving the implementation itself by providing better integration of the tools for drawing and inspecting strategies, as well as adapting the code to work with other underlying theorem provers.

An additional motivation for the language has been as a language for learning high-level proof strategies from exemplar proofs, first motivated in [6]. With partners on the AI4FM project ([www.ai4fm.org](http://www.ai4fm.org)) we are working to support this form of *analogical reasoning*. In particular, we are working on extracting strategy graphs from Isabelle proofs, which we then generalise by graph rewriting and transformations. Automatically extracting new goal types is an important part of this process, and are also working on using the partial order for the goal types in order to support sub-typing.

**Acknowledgements** Several of the ideas behind the language is joint with Lucas Dixon, while Andrius Velykis has been very supportive with Eclipse. Also thanks to Alex Merry, Rod Burstall, Ewen Maclean, Alan Bundy, Cliff Jones, Andrew Ireland, Leo Freitas, Yuhui Lin, and Colin Farquhar for suggestions and discussions. This work has been supported by EPSRC grants: EP/H023852, EP/H024204 and EP/J001058, the John Templeton Foundation, and the Office of Navel Research.

## References

1. A. Asperti, W. Ricciotti, C. Sacerdoti, and C. Tassi. A new type for tactics. In *PLMMS'09*, pages 229–232, 2009.
2. David Aspinall, Ewen Denney, and Christoph Lüth. A tactic language for hiproofs. In *MKM'08*, pages 339–354, Berlin, Heidelberg, 2008. Springer-Verlag.
3. Serge Autexier and Dominik Dietrich. A tactic language for declarative proofs. In *ITP'10*, volume 6172 of *LNCS*, pages 99–114. Springer, 7 2010.
4. A. Bundy. The use of explicit plans to guide inductive proofs. pages 111–120. Springer-Verlag, 1988.
5. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
6. A. Bundy, G. Grov, and C.B. Jones. Learning from experts to aid the automation of proof search. In *PreProc of AVoCS'09*, pages 229–232, 2009.
7. Lucas Dixon and Jacques D. Fleuriot. Isaplanner: A prototype proof planner in isabelle. In *CADE-19*, volume 2741 of *LNCS*, pages 279–283. Springer, 2003.
8. Lucas Dixon and Aleks Kissinger. Open graphs and monoidal theories. *CoRR*, abs/1011.4114, 2010.
9. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. Springer, 2006.
10. M. Giero, F. Wiedijk, and Mariusz Giero. MMode, a mizar mode for the proof assistant coq. Technical report, January 07 2004.
11. John Harrison. A mizar mode for HOL. In *TPHOLs*, volume 1125 of *LNCS*, pages 203–220. Springer, 1996.
12. Mateja Jamnik. *Mathematical Reasoning with Diagrams: From Intuition to Automation*. CSLI Press, Stanford University, 2001.
13. Andre Joyal and Ross Street. The geometry of tensor calculus I. *Advances in Mathematics*, 88:55–113, 1991.
14. A. Kissinger, A. Merry, L. Dixon, R. Duncan, M. Soloviev, and B. Frot. Quantomatic. <https://sites.google.com/site/quantomatic/>, 2011.
15. Aleks Kissinger, Alex Merry, and Matvey Soloviev. Pattern graph rewrite systems. *CoRR*, abs/1204.6695, 2012.
16. Roger Penrose. Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications*, pages 221–244. Academic Press, 1971.
17. I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. Towards formal proof script refactoring. In *CICM'11*, volume 6824 of *LNCS*, pages 260–275. Springer, 2011.